Contracts for Message-Passing Programs

Extended Abstract

Ziqing Luo and Stephen F. Siegel (Advisor)
Verified Software Laboratory, Department of Computer and Information Sciences, University of Delaware
Newark, Delaware, USA 19716
(ziqing|siegel)@udel.edu

ABSTRACT

Verification of message-passing parallel programs is challenging because of the state-explosion problem. A procedure-level contract system for parallel programs can help overcome this challenge by allowing procedures to be verified individually. A contract system is used to specify intended behaviors of programs. Contracts can be checked at run-time or verified formally. There is a mature theory of contracts for sequential programs, but little work has been done on parallel programs, and even less for message-passing parallel programs. We have developed a theory of contracts for message-passing programs and realize this theory in a contract language for programs that use the Message Passing Interface (MPI). We are also developing a verification tool that uses symbolic execution and model checking techniques to prove that MPI programs satisfy their contracts.

1 INTRODUCTION

Message-passing is a pervasive parallel programming model, particularly in high performance and scientific computing. Correctness of programs in this area is paramount, but is difficult to guarantee using traditional approaches such as manual inspection and testing. In large part, this is due to the size and complexity of the programs, and from nondeterministic behavior introduced by parallelism. For the same reasons, standard automated verification approaches, such as monolithic model checking, do not scale to realistic scientific programs.

Divide and conquer is a fundamental software engineering strategy which entails decomposing a large problem into smaller tasks. The *Design by Contract* [7] methodology is a well-known example of this approach. By assigning a *contract* specifying intended behavior to each procedure of a large system, ensuring the correctness of the system is decomposed to ensuring that each procedure satisfies its contract.

A procedure contract includes a *precondition* and *postcondition*. For example, the precondition might state that a formal parameter x must be positive and the postcondition might state that the function returns $\ln(x)$. The contract encodes the claim: *if the precondition holds when the procedure is called, the postcondition will hold when the procedure returns.* Contracts can be checked at runtime using assertions. If a precondition is violated, the caller has a defect. If the precondition held but the postcondition is violated, the procedure was implemented incorrectly.

A contract system also enables modular verification. To verify that a program is correct, each procedure p can be verified independently. When verifying p, one assumes that all procedures q called by p satisfy their contracts. So only the contract of q (and not its implementation) is used when verifying p. If one verifies every procedure in this way, the program is correct.

Contract systems and verification tools addressing sequential programs have been developed for many programming languages [2, 3, 5, 6]. A few contract systems have also been extended to deal with shared memory concurrent programs [4, 8, 9] but, to the best of our knowledge, no one has extended these ideas to message-passing parallel programs. Shared memory and message-passing concurrent programs have very different characteristics; techniques used to verify one of them are not necessarily suitable for the other.

The remainder of this document consists of a summary of the theory (§2) and language (§3) for message-passing contracts, a short description of the verification system (§4) that we are currently building, and the discussion of future work (§5).

2 A THEORY FOR MESSAGE-PASSING CONTRACTS

The semantics of sequential contracts is inappropriate for message-passing programs for two reasons: First, the behavior of a process executing a procedure depends on the actions of other processes. Second, the preconditions and postconditions required to specify a parallel program need to relate the state on different processes in subtle ways. A state in which all processes are at desired locations will not necessarily exist during an execution.

We dealt with the first problem by avoiding it. Instead of trying to specify the behavior of a single process executing an arbitrary procedure, We specify the behavior of a whole set of processes executing a *collective-style* procedure. A collective-style procedure is one that is intended to be called by all processes in a communicator, and all messages "stay within" the procedure—i.e., every message received by a process in the procedure was sent by another process in the procedure, and every message sent by a process in the procedure is received by another process in the procedure. The (blocking) MPI collective functions are all examples of collective-style procedures, but so are many other user-defined functions. For example, the "main" function of the typical MPI program (ignoring the MPI initialization and finalization) is collective-style.

Collective-style procedures may call non-collective-style procedures. That is fine—our approach can still be used to specify and verify such collective-style procedures, it just won't be able to decompose the verification problem for them into smaller pieces.

For the second problem, We adapted an idea from *collective assertions* [11]. The idea is to allow the user to specify assertions in an intuitive "bulk-synchronous" style even when executions are not bulk-synchronous. Behaviors of collective-style functions can be specified in the same way: one can "pretend" as if all processes are calling the function simultaneously when specifying the behavior.

`\on(`*e*`,` *rank*`)`	the value of expression *e* on the process of rank *rank*
`\mpi_comm_size`	a constant denoting the number of processes in the communicator
`\mpi_comm_rank`	a constant denoting the rank of this process in the communicator
`\mpi_extent(`*datatype*`)`	the extent of an MPI datatype
`\mpi_offset(`*buf*`,` *count*`,` *datatype*`)`	a pointer value: `(char*)`*buf* `+` *count*`*\mpi_extent(`*datatype*`)`
`\mpi_region(`*buf*`,` *count*`,` *datatype*`)`	the memory region containing a sequence of *count* objects of a C type corresponding to *datatype* whose first element is pointed to by *buf*
`\mpi_agree(`*e*`)`	a predicate asserting that all processes have the same value for expression *e*
`\mpi_valid(`*buf*`,` *count*`,` *datatype*`)`	a predicate asserting that the pointer *buf* points to the first element of a sequence of *count* objects of a C type corresponding to *datatype*
`\mpi_equals(`*region0*`,` *region1*`)`	a predicate asserting that two memory regions hold the same sequence of values
`waitsfor` *R*`;`	a contract clause expressing that a process cannot exit the function until all processes listed in *R* have entered the function

Figure 1: New contract primitives added to ACSL

A contract then will be interpreted in two *collective states*: a collective pre-state and a collective post-state. Conceptually, a collective state is formed by taking a *snapshot* of each process state as control passes through a certain location. The snapshots are then joined together to form the collective state. Process states in a collective pre-state all have their controls at the entry of a function and ones in a collective post-state all have their controls at the exit of a function.

3 COLLECTIVE CONTRACT LANGUAGE

We did not design a new contract language from scratch, but extended an existing sequential contract language to deal with message-passing programs. The ANSI/ISO C Specification Language (ACSL) [1] is an annotation-based specification language for C programs. It can be used to specify function contracts as well as other concepts, such as loop invariants.

Our collective contract language extends ACSL with the small set of new constructs listed in Figure 1. The language adopts the contract semantics for collective-style functions described in §2.

Figure 2 shows an example of collective contracts. Note that in this example, the `\mpi_region`s can be simply replaced with `u[0]` and `u[nxl+1]`, and the `\mpi_valid` can be replaced with the regular ACSL `\valid`. These MPI specific primitives are designed for more general cases in which message buffer pointers have type `void*`.

The semantics of the collective contract language relaxes the one borrowed from collective assertions (described in §2) in order to allow formulas to evaluate on incomplete collective states. Not every participating process has a snapshot in an incomplete collective state. This is because the collective contract language provides the `waitsfor` clause to explicitly state synchronization dependencies among processes. A process can evaluate a collective contract as long as the collective state contains all snapshots of processes listed in the `waitsfor` clause.

4 A VERIFICATION SYSTEM FOR MPI COLLECTIVE-STYLE FUNCTIONS

We are working on an automatic verification system, which verifies that a collective function satisfies a function contract. The function contract is written in the language introduced in §3.

```
double* u; int left, right, nxl;
...
/*@\mpi_collective(MPI_COMM_WORLD, P2P):
  @   requires nxl > 0 && \mpi_valid(u, nxl + 2, MPI_DOUBLE);
  @   requires left != \mpi_comm_rank &&
  @            right != \mpi_comm_rank;
  @   requires left == MPI_PROC_NULL ||
  @                   (0 <= left && left < \mpi_comm_size);
  @   requires right == MPI_PROC_NULL ||
  @                   (0 <= right && right < \mpi_comm_size);
  @   ensures \mpi_valid(u, nxl + 2, MPI_DOUBLE);
  @   behavior hasLeftNeighbor:
  @     assumes left != MPI_PROC_NULL;
  @     requires \mpi_comm_rank == \on(left, right);
  @     assigns \mpi_region(u, 1, MPI_DOUBLE);
  @     ensures u[0] == \on(left, u[nxl]);
  @     waitsfor left;
  @   behavior hasRightNeighbor:
  @     assumes right != MPI_PROC_NULL;
  @     requires \mpi_comm_rank == \on(right, left);
  @     assigns \mpi_region(&u[nxl+1], 1, MPI_DOUBLE);
  @     ensures u[nxl + 1] == \on(right, u[1]);
  @     waitsfor right;
  @*/
void exchange_ghost_cells() {...}
```

Figure 2: Collective contract for a ghost cell exchange function.

The verification system will be built on top of the CIVL verification framework [10]. CIVL consists of an intermediate verification language CIVL-C and a verifier for CIVL-C programs. CIVL-C provides a group of primitives and libraries dealing with concurrency and specifications, e.g., one can use `$assume(expr)` to ignore the current execution unless `expr` holds. The verifier uses model checking and symbolic execution techniques.

Given a C/MPI program annotated with contracts and a function f in that program, the new system will produce a CIVL-C program containing collective assertions that hold if and only if f satisfies its contract. The body of an annotated function called by f will be replaced with code generated according to its contract.

5 FUTURE WORK

In addition to the implementation of the verification system, we plan to extend the message-passing contract theory and language to deal with more aspects of MPI programs, such as non-collective-style functions, non-blocking communication and one-sided communication.

REFERENCES

[1] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. 2008. ACSL: ANSI C Specification Language. https://frama-c.com/acsl.html. (2008).

[2] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. 2006. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects (FMCO'05)*. Springer-Verlag, Berlin, Heidelberg, 342–363. https://doi.org/10.1007/11804192_16

[3] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C. In *Software Engineering and Formal Methods*. Springer, Berlin, Heidelberg, 233–247. https://doi.org/10.1007/978-3-642-33826-7_16

[4] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. 2009. VCC: Contract-based modular verification of concurrent C. In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. IEEE Computer Society, 429–430. https://doi.org/10.1109/ICSE-COMPANION.2009.5071046

[5] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370. http://dl.acm.org/citation.cfm?id=1939141.1939161

[6] B Meyer. 1987. Eiffel: Programming for Reusability and Extendibility. *SIGPLAN Not.* 22, 2 (Feb. 1987), 85–94. https://doi.org/10.1145/24686.24694

[7] Bertrand Meyer. 1992. Applying 'Design by Contract'. *Computer* 25, 10 (Oct 1992), 40–51. https://doi.org/10.1109/2.161279

[8] Piotr Nienaltowski, Bertrand Meyer, and Jonathan S. Ostroff. 2006. Contracts for concurrency. *Report-University of York Department of Computer Science YCS* 405 (2006), 27. https://doi.org/10.1007/s00165-007-0063-2

[9] Edwin Rodríguez, Matthew Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, et al. 2005. Extending JML for modular specification and verification of multi-threaded programs. In *ECOOP 2005-Object-Oriented Programming*. Springer, Berlin, Heidelberg, 551–576. https://doi.org/10.1007/11531142_24

[10] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. 2015. CIVL: The Concurrency Intermediate Verification Language. In *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis, Proceedings (SC '15)*. IEEE Press, Piscataway, NJ, USA, 61:1–61:12. https://doi.org/10.1145/2807591.2807635

[11] Stephen F Siegel and Timothy K Zirkel. 2011. Collective assertions. In *Verification, Model Checking, and Abstract Interpretation*. Springer-Verlag, Berlin, Heidelberg, 387–402. http://dl.acm.org/citation.cfm?id=1946284.1946311